Design Patterns:

==========

Design Pattern is a solution to a commonly occurring problem in software design.

Design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable.

Christopher Alexander was the first person who invented all the above Design Patterns in 1977.

But later the Gang of Four - Design patterns, elements of reusable object-oriented software book was written by a group of four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1995.

That's why all the above 23 Design Patterns are known as Gang of Four (GoF) Design Patterns.

Why we need Design Patterns:

==========

Problem:

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution:

Singleton design pattern is the best solution of above specific problem. So, every design pattern has some specification or set of rules for solving the problems.

Advantages of Design Patterns:

==========

1. They are reusable in multiple projects.

2. They provide the solutions that help to define the system architecture.

3. They provide transparency to the design of an application.

4. Code reusability

5. Speed development.

6. Easy to test the apps.

When to use the Design Patterns:

==========

We must use the design patterns during the analysis and requirement phase of SDLC(Software Development Life Cycle).

Categories of Design Patterns

==========

Basically, design patterns are categorized into two parts:

Core Java (or JSE) Design Patterns.

JEE Design Patterns.

Core Java Design Patterns:

---

1. Creational Design Patterns

2. Structural Design Patterns

3. Behavioural Patterns

Creational Design Patterns:

==========

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

Types of creational design patterns:

---

1. Factory Method Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern
6. Object Pool Pattern

Factory Method Pattern:

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

Advantage of Factory Design Pattern:

---

1. Factory Method Pattern allows the sub-classes to choose the type of objects to create.

2. It promotes the loose-coupling.

Example:

Calculate Electricity Bill : A Real World Example of Factory Method

```
abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}//end of Plan class.

class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
           rate=3.50;
       }
```

```
  }//end of DomesticPlan class.


 class  CommercialPlan extends Plan{
  //@override
   public void getRate(){
      rate=7.50;
  }
```

Abstract Factory Pattern:

Abstract Factory Pattern says that just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

Advantage of Abstract Factory Pattern:

Abstract Factory Pattern isolates the client code from concrete (implementation) classes.

It eases the exchanging of object families.

It promotes consistency among objects.

Example:

```
interface CurrencyType{
String getCurrencyType();
}


class India implements CurrencyType{
public Sting getCurrencyType(){
return "INR"
```

}

}

Singleton design pattern:

Singleton Pattern says that just"define a class that has only one instance and provides a global point of access to it".

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern:

Early Instantiation: creation of instance at load time.

Lazy Instantiation: creation of instance when required.

Advantage of Singleton design pattern:

Saves memory because object is not created at each request. Only single instance is reused again and again.

How to create Singleton design pattern?

To create the singleton class, we need to have static member of class, private constructor and static factory method.

Static member: It gets memory only once because of static, it contains the instance of the Singleton class.

Private constructor: It will prevent to instantiate the Singleton class from outside the class.

Static factory method: This provides the global point of access to the Singleton object and returns the instance to the caller.

Understanding early Instantiation of Singleton Pattern:

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

```
class A{
 private static A obj=new A();//Early, instance will be created at load time
 private A(){}

 public static A getA(){
  return obj;
 }

 }
```

Understanding lazy Instantiation of Singleton Pattern:

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

```
class A{
 private static A obj;
 private A(){}
 public static A getA(){
   if (obj == null){
     synchronized(Singleton.class){
       if (obj == null){
          obj = new Singleton();//instance will be created at request time
       }
    }
    }
  return obj;
 }
 }
```

Prototype Design Pattern:

Prototype Pattern says that cloning of an existing object instead of creating new one and can also be customized as per the requirement.

```
interface Prototype {
    public Prototype getClone();
}

class EmployeeRecord implements Prototype{
@Override
   public Prototype getClone() {

      return new EmployeeRecord(id,name,designation,salary,address);
   }
}
```

Builder Design Pattern:

Builder Pattern says that "construct a complex object from simple objects using step-by-step approach"

It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

Example:

Lets take an bank object its having different account types(Savings,Checking, MM, Personal Loan and Student Loans) and we need to maintain the balance history as well as account status. If we want to craee the objects with these many fields its diffucult to maintain them. In this case Builder Design Pattern will hep us to build the complex objects in single step.

Object Pool Pattern:

Mostly, performance is the key issue during the software development and the object creation, which may be a costly step.

Object Pool Pattern says that " to reuse the object that are expensive to create".

Basically, an Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back. Objects in the pool have a lifecycle: creation, validation and destroy.

Structural Design Patterns:

==========

The structural design patterns simplifies the structure by identifying the relationships.

Structural design patterns are concerned with how classes and objects can be composed

Types of structural design patterns:

Adapter Pattern:

Adapter pattern works as a bridge between two incompatible interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

Example:

A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Bridge Pattern:

Its decouples implementation class and abstract class by providing a bridge structure between them.

Example:

Whenever we purchase any smartphone, we get a charger. The charger cable now-a-days can be separated, so we can use it to connect as USB cable to connect other devices.

Composite Pattern:

Allowing clients to operate on hierarchy of objects.

There are six people on my contact list: Mom, Dad, Uncle Bob, Cousin Nick, Aunt Julia, and Amy Kinser. The Parents group is composed of Mom and Dad, whereas the Uncle Bob's Family group is composed of Uncle Bob, Cousin Nick, and Aunt Julia.

In this example, Contacts is represented in a tree structure made of nodes. A node is either a group of people or a single person. If a node is a group of people, like Parents, then it contains other nodes. If a node is a single person, like Mom, then it's a leaf node.

Decorator Pattern:

Adding functionality to an object dynamically.

Example:

Icecream is a classic example for decorator design pattern. You create a basic icecream and then add toppings to it as you prefer. The added toppings change the taste of the basic icecream. You can add as many topping as you want.

Facade Pattern:

Providing an interface to a set of interfaces. Facade design pattern is one among the other design patterns that promote loose coupling. By hiding the complexity behind it and exposing a simple interface it achieves abstraction

Example:

Lets take a car, starting a car involves multiple steps. Imagine how it would be if you had to adjust n number of valves and controllers. The facade you have got is just a key hole. On turn of a key it send

instruction to multiple subsystems and executes a sequence of operation and completes the objective. All you know is a key turn which acts as a facade and simplifies your job.

Proxy Pattern:

Representing another object.

Simply, proxy means an object representing another object.

According to GoF, a Proxy Pattern "provides the control for accessing the original object".

Example:

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that's exactly what the Proxy pattern does .

Behavioral Design Patterns:

==========

Behavioral design patterns are concerned with the interaction and responsibility of objects.

In these design patterns,the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies..

There are 12 types of structural design patterns:

- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern

- Visitor Pattern
- Null Object

Chain Of Responsibility Pattern:

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

Example:

A real world example for the chain of responsibility is the chain of command in a company. For example if an employee needs an approval for a task, he gives the report to his manager. If the manager can't approve the report, for example because the costs surpass his authority, he gives the report to his manager until a manger with enough authority is found or until the report is rejected.

Command Pattern:

A Command Pattern says that "encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command".

Example:

One example of the command pattern being executed in the real world is the idea of a table order at a restaurant: the waiter takes the order, which is a command from the customer.This order is then queued for the kitchen staff. The waiter tells the chef that the a new order has come in, and the chef has enough information to cook the meal.

Interpreter Pattern:

Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

The best example of interpreter design pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

SQL Parsing uses interpreter design pattern.

Iterator Pattern:

To access the elements of an aggregate object sequentially without exposing its underlying implementation.

The Iterator pattern is also known as Cursor.

In collection framework, we are now using Iterator that is preferred over Enumeration.

Mediator Pattern:

To define an object that encapsulates how a set of objects interact.

I will explain the Mediator pattern by considering a problem. When we begin with development, we have a few classes and these classes interact with each other producing results. Now, consider slowly, the logic becomes more complex when functionality increases. Then what happens? We add more classes and they still interact with each other but it gets really difficult to maintain this code now. So, Mediator pattern takes care of this problem.

Example:

A great real world example of mediator pattern is traffic control room at airports. If all flights will have to interact with each other for finding which flight is going to land next, it will create a big mess.

Rather flights only send their status to the tower. These towers in turn send the signals to conform which airplane can take-off or land. We must note that these towers do not control the whole flight. They only enforce constraints in the terminal areas.

Memento Pattern:

A Memento Pattern says that "to restore the state of an object to its previous state". But it must do this without violating Encapsulation.

The Memento pattern is also known as Token.

Example:

Undo or backspace or ctrl+z is one of the most used operation in an editor. Memento design pattern is used to implement the undo operation. This is done by saving the current state of the object as it changes state.

Observer Pattern:

An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

Example:

A real world example of observer pattern can be any social media platform such as Facebook or twitter. When a person updates his status – all his followers gets the notification.

A follower can follow or unfollow another person at any point of time. Once unfollowed, person will not get the notifications from subject in future.

State Pattern:

A State Pattern says that "the class behavior changes based on its state". In State Pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

The State Pattern is also known as Objects for States.

Example:

To make things simple, let's visualize a TV box operated with remote controller. We can change the state of TV by pressing buttons on remote. But the state of TV will change or not, it depends on the current state of the TV. If TV is ON, we can switch it OFF, mute or change aspects and source. But if TV is OFF, nothing will happen when we press remote buttons.

Strategy Pattern:

A Strategy Pattern says that "defines a family of functionality, encapsulate each one, and make them interchangeable".

The Strategy Pattern is also known as Policy.

Example:

I have a project where the users can assign products to people in a database. This assignment of a product to a person has a status which is either "Approved" or "Declined", which is dependent on some business rules. For example: if a user assigns a product to a person with a certain age, it's status should be declined; If the difference between two fields in the item is larger than 50, it's status is declined, etc.

Now, at the moment of development these business rules are not yet all completely clear, and new rules could come up at any time.

Template Pattern:

A Template Pattern says that "just define the skeleton of a function in an operation, deferring some steps to its subclasses"

Example:

Suppose you want to prepare Coffee (let say BruCoffee). Then you need to follow some steps or procedures such as Boil Water, Add Milk, Add Sugar, and Add BruCoffee

Visitor Pattern:

---

it's used to manage algorithms, relationships and responsibilities between objects.

Examples:

One example I have seen for the Visitor pattern in action is a taxi example, where the customer calls orders a taxi, which arrives at his door. Once the person sits in, the visiting taxi is in control of the transport for that person.

Shopping in the supermarket is another common example, where the shopping cart is your set of elements. When you get to the checkout, the cashier acts as a visitor, taking the disparate set of elements (your shopping), some with prices and others that need to be weighed, in order to provide you with a total.

//

JEE or J2EE Design Patterns

==========

J2EE design patterns are built for the developing the Enterprise Web-based Applications.

In J2EE , there are mainly three types of design patterns, which are further divided into their sub-parts:

1. Presentation Layer Design Pattern

==========

Intercepting Filter Pattern

It is used to provide interception and manipulation of requests as well as response prior to and preceding the processing of the request.

Front Controller Pattern:

A centralized access point allows for non-duplication of the control logic needed to handle a request. Front Controller is to handle such request by acting as an initial point.

View Helper Pattern:

It is used to provide a different view, hiding the logic present in the code. Now the logic and the view are completely independent to provide ease for developers and designers.

Composite View Pattern:

Small sub views can be created using the composite view. These sub views can be integrated to create a singular view.

2. Business Layer Design Pattern

==========

Business Delegate Pattern:

The business delegate pattern is one of the Java EE design patterns. It is used in order to decouple or reduce the coupling between the presentation tier and business services. It is also required to hide the details of implementation of the services, meaning it is needed to remove the function of lookup in the business tier code within the presentation tier code.

Service Locator Pattern:

The design pattern, service locator is an important part in software development. Looking up for a service is one of the core features of service locator. A robust abstraction layer performs this function. The design pattern uses a central registry called Service Locator.When it is required to locate a number of services, through JNDI lookup, the service locator pattern is used.

Session Facade Pattern:

The session façade pattern's core application is development of enterprise apps. You can also call it a logical extension of GoF designs. The pattern encases the interactions which are happening between the low-level components, which is Entity EJB. It is implemented as a higher level component, Session EJB. After which, it is responsible for providing a common and an only interface in order for the app to function or a part of the app

Transfer Object Pattern:

It is one of the Java EE design patterns. We need Transfer Object when we need to pass the data across various attributes in a packet to the server. Value Object is another name for transfer object. The transfer object is just a class of POJO which has a method of the getter and setter. It is serializable which means we can transfer it through the network.

3. Integration Layer Design Pattern

==========

Data Access Object Pattern:

The data access object in a computer software which is as an object which is responsible for providing abstract interface for communication to a specific form of database. Through the method of mapping, the app is able to call the persistence layer and the DAO then provides a certain type of data operations. You don't need to expose what the database actually contains. This segregation is able to support the Single responsibility principle. It splits the need for the app in terms of data access from how can these

needs be fulfilled with certain database schema, DBMS, etc. These needs can be domain specific or data types which is the public interface of the data access object.

Web Service Broker Pattern:

The web service broker uses web protocols and XML. We can use this pattern to expose and broker the services. Assume a circumstance, where multiple organizations are lined up in order to request info from a number of service providers. A broker provides the central medium which makes the transfer of information happen.

It is a general gateway or an address in order for the client apps to be able to access a large, diverse variety of services. These services might need a broker in order to make an interaction with either only a single server app or multiple server apps. The broker performs certain tasks. It is responsible for receiving SOAP requests from the client apps in XML format along with authenticating the request and checking for the authorization. You can also use it to generate calls to multiple server apps, which depends on nature of the request.

Questions:

1. **Explain Design patterns solid principles?**

   Single Responsibility: One class should have one and only one responsibility.

   Example: Person class can have persons related data and Account must have account related data.

   Open Close Principle: Software components should be open for extension, but closed for modification. It means that the application classes should be designed in such a way that whenever fellow developers want to change the flow of control in specific conditions in application, all they need to extend the class and override some functions and that's it.

   Example: For example, spring framework has class DispatcherServlet. This class acts as a front controller for String based web applications. To use this class, we are not required to modify this class. All we need is to pass initialization parameters and we can extend its functionality the way we want.

   Interface Segregation Principle: Clients should not be forced to implement unnecessary methods which they will not use

   Example: Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf(). Now client 'A' wants to use this interface but he

intends to use reports only in PDF format and not in excel. Will he be able to use the functionality easily?

Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to users to use only the required functionality only.

Dependency Inversion Principle: We should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

Example: In the spring framework, all modules are provided as separate components which can work together by simply injecting dependencies in other modules. This dependency is managed externally in XML files.